

AUTONOMOUS MEMORY CHECKER FOR RUNTIME SECURITY ASSURANCE
AND METHOD THEREFORE

TECHNICAL FIELD OF THE INVENTION

[0001] The present invention generally relates to security assurance of software, and more particularly relates to a hardware method for adding security assurance to runtime software.

BACKGROUND OF THE INVENTION

[0002] Assuring code integrity of embedded systems software at runtime is becoming a significant security issue for an embedded electronic device. An example of high volume electronic devices where it is beneficial to ensure the integrity of the software stored in memory are personal digital assistants (PDAs) and cell phones. In general, these devices use internal microprocessors to execute instructions in two stages: boot-time and runtime. A trusted computer system uses cryptographic software to authenticate each stage prior to its execution.

[0003] During boot-time, an embedded processor executes fundamental hardware and software initialization instructions (the boot code) stored in non-volatile memory such as electrically erasable read only memory (such as a flash non-volatile memory), read only memory (ROM), or electrically programmable read only memory (EPROM). The purpose of the boot code is to validate and configure the hardware and hardware related data in order to present a known execution environment and user interface for the system software. For trusted computer operation, the boot code also validates or authenticates the program application and operating system (OS) memory at boot-time to ensure that it contains the code that is expected and hence trusted. Once the boot-time instruction has checked for authenticity, the system control is passed to the validated OS/application execution image and enters the runtime mode.

[0004] During runtime, the processor executes code out of the OS/application image. The OS will typically establish a multi-process runtime execution environment and load any start-up or embedded applications in preparation for normal runtime processes. It is during

this later time, when the runtime environment is fully established, that it is possible for untrusted user application code or downloaded dynamic code to be run and for memory corruption to occur on the original trusted and authenticated code, due to non-boot-time factors such as computer viruses or internal programming bugs. Furthermore, the OS/application program memory validated at boot-time could maliciously be replaced by a runtime memory, which is completely different from what is authenticated at boot-time. This is known as a "piggy back" attack, which poses a considerable security threat to embedded system software. Requiring physical access to the product platform, a runtime memory device is piggy-backed over the original validated memory device. It completely replaces the validated boot device after the boot process finishes and passes control to the external memory, hence creating a completely different, unchecked runtime environment.

[0005] Conventional methods of validating instruction codes in systems software only authenticate the instruction memory once. One-time validation is performed either by generating a one-way memory reference signature or asymmetric key digital signature using a cryptographic algorithm. Prior to execution, a message authentication code or a similar digital signature is generated on the instruction memory. If the generated result matches a pre-determined value stored elsewhere in the system or within the encrypted asymmetric digital signature, the instructions are permitted to execute. However, current solutions do not address the risk of tampering with the instruction memory contents after the initial authentication. During runtime, instruction codes are vulnerable to piggy-back or software attacks which will replace the authenticated instruction memory or data. Integrity of initially authenticated codes can be compromised by physical replacement of memory or diversion of processor to execute code from a memory range that was not originally authenticated. For example, a computer virus can remain dormant during boot-time and then cause an exception in the processor to make it execute a different set of instructions.

[0006] A large portion of the system software controlling an electronic device should not change during the course of operation of the device. Even though the system software is loaded into memory with high assurance by initial authentication, the probability of system degradation goes up after several million clock cycles and continued exposure to un-trusted code. If the system is corrupted, it cannot be trusted to perform self-check functions. In fact, the reference data/memory that is used to make these self-checks are security targets or vulnerabilities and examples of data that is security sensitive and should never change. Other security sensitive data could be configuration registers, which control internal data

visibility external to an embedded device or OS sensitive data such as (interrupt service routine (ISR) addresses or specific memory management unit (MMU) page tables or specific file system data such as user and group identifiers and passwords.

[0007] Furthermore, many embedded systems such as cell phones and PDA's enter the boot mode only if they are disconnected from power sources. Therefore, the integrity of original instruction codes of an embedded device in the runtime mode confronts a gradually increasing risk as the device runs downloaded application programs and communicates with external devices. The magnitude of this risk is therefore especially significant in devices that do not normally get disconnected from their power sources and that do not reboot to authenticate until power failures occur. Moreover, operating systems and application software are stored in different regions of memory. The fact that the software is stored in different regions and the software may be received from a number of sources, some reputable, others that cannot be substantiated poses a long term tracking and security problem.

[0008] Accordingly, it is desirable to devise a memory checker for runtime security assurance of software that is loaded to memory during the boot-time. It would be of benefit if the memory checker was autonomous to ensure checks occur. It would be of further benefit if the device was energy efficient and did not significantly impact runtime system performance by requiring frequent bus accesses.

BRIEF DESCRIPTION OF THE DRAWINGS

[0009] The present invention will hereinafter be described in conjunction with the following drawing figures, wherein like numerals denote like elements, and

[0010] FIG. 1 is a block diagram of an autonomous memory checker in accordance with one embodiment of the present invention;

[0011] FIG. 2 is a first flow chart illustrating operating steps of an autonomous memory checker for runtime security assurance in accordance with one embodiment of the present invention;

[0012] FIG. 3 is a system block diagram showing an autonomous integrity checker coupled to a host processor and memory blocks; and

[0013] FIG. 4 is a second flow chart illustrating operating steps of an autonomous memory checker for runtime security assurance in accordance with one embodiment of the present invention.

DETAILED DESCRIPTION OF THE INVENTION

[0014] The following detailed description is merely exemplary in nature and is not intended to limit the invention or the application and uses of the invention. Furthermore, there is no intention to be bound by any expressed or implied theory presented in the preceding technical field, background, brief summary or the following detailed description.

[0015] FIG 1 is a block diagram of an autonomous memory checker 10 in accordance with one embodiment of the present invention. A host processor (not shown) first sets programmable parameters such as memory start addresses, lengths, direct memory access (DMA) read frequency, maximum burst size, and memory block enables. The host then sends a command via host bus 50 to a controller 25 of autonomous memory checker 10 to start an initial memory reference mode. In the initial memory reference mode, controller 25 requests a DMA controller 15 to fetch the memory contents via a DMA master bus 45. The memory content is trusted information. An authentication engine 20 receives the memory contents via an internal bus 55 and generates appropriate memory references. The authentication engine 20 can generate memory reference values such as hash functions for large blocks of data, a simple copy for small amounts of data, or cyclic redundancy check (CRC) for fast processing of large amounts of data. The memory reference values output from the authentication engine 20 are then stored in a memory reference file 40 via internal buses 80 and 85. Memory reference file 40 stores the memory reference values in memory during the boot-time mode. These memory reference values are repeatedly compared against runtime reference values generated during a runtime mode to ensure the integrity of the trusted information, which should not have changed since the boot-time mode.

[0016] An example of trusted information is software critical to the operation of an electronic device. The manufacturer loads the software in the electronic device. The

software is trusted information since the manufacturer or other trusted entity has provided the software to the electronic device. In general, prior art electronic devices only check the integrity of the trusted information during boot-time mode. The trusted information is prone or susceptible to modification without the knowledge of the user of the electronic device. Modification of the trusted information by an outside agency without the knowledge of the user could have severe consequences. Checking the trusted information using a software program is self defeating because it is the integrity of the information in memory that is at question. In other words, the software checking program could be modified similar to the trusted information. It should be noted that the trusted information can be any type of information that should not be changed or modified and is not limited to software.

[0017] During runtime, autonomous memory checker 10 can proactively monitor the authenticity of validated memory contents. It can fetch values from memory and perform a variety of monitoring schemes. One possible method is "point checks," which retrieves data from one isolated location of the memory. Another method involves "wandering checks," which reads a value from a location and compares it with the previously stored value. If the two values match, the pointer is incremented and a new value is read and stored. This stored value is later compared with a value reread from the same memory location. The wandering checks repeat themselves within the boundaries of the allocated space.

[0018] Another common scheme is "block checks." This method is often called "hashing" or CRC. Block checks perform a compression-type function on a specified block of data and store a distinct representative value (i.e. output of a discrete mathematical function) for later comparison.

[0019] Other monitoring functions include write-read checks, active wait-for checks, and software to hardware handoff checks. The write-read checks load a random value into a memory location and then retrieve it later. Any change in the stored value indicates possible data corruption. The write-read method requires DMA controller 15 to be capable of both reading and writing to desired memory locations.

[0020] The active wait-for checks require the system software to "check-in" periodically for comparison. Lastly, the software to hardware handoff checks prevent buffer overflow attacks which occur in a subroutine where the space allocated for a buffer is smaller than the number of words written into it. The overflowing words fall into the stack space allocated

for critical data such as program counter. When the subroutine returns, the program counter will be incorrect. One way to protect against the buffer overflow attack is to command healthy software to initiate a hardware check in a secure state. The healthy software informs the hardware that it is about to allocate a buffer in a subroutine and that location of the critical data is at a particular address. The hardware caches the critical values. Shortly after, the hardware looks to see if the value at the edge of the buffer did not change. Although the software to hardware handoff checks involve some overhead, it significantly adds security assurance to highly sensitive data.

[0021] Autonomous memory checker 10 stops its operation and asserts an appropriate interrupt to signal potential degradation of validated memory contents if a reference mismatch occurs, address/length error, or watchdog time-out occurs during runtime. It should be noted that autonomous memory checker 10 operates independently and cannot be deactivated by other logic blocks or programs during its operation. This ensures that trusted information will always be checked during runtime mode. Any error that occurs during runtime will cause the autonomous memory checker 10 to halt and signal an error to the host. In an embodiment of the system, a hardware reset is required to exit the error state.

[0022] Autonomous memory checker 10 also includes a timer module 35 and a clock control 30. Timer module 35 deactivates certain logic blocks of autonomous memory checker 10 in various stages of its operation to provide energy savings. Timer module 35 is on continuously and in part, controls the operation of clock control 30, controller 25, authentication engine 20, and other blocks. Timer module 35 contains a bus watchdog timer that ensures that DMA controller 15 is granted bus access within an acceptable amount of time. Clock control 30 provides clock signals to controller 25, authentication engine 20, and memory reference file 40 if clock control 30 is in an active mode.

[0023] DMA controller 15 allows the autonomous memory checker 10 to be a bus master. Autonomous memory checker 10 can fetch contents of the memory without requesting permission to the host at any time. DMA controller 15 has read capability. The direct memory access capability of DMA controller 15 allows autonomous memory checker 10 to be reliable in the event that the host processor cannot be trusted due to software corruption.

[0024] FIG 2 is a first flow chart demonstrating conceptual operating steps for the autonomous memory checker 10 of FIG. 1. During the boot-time after reset, the

autonomous memory checker 10 receives a signal to enter the initial memory reference mode 110 from the host. In the initial memory reference mode 110, autonomous memory checker 10 fetches specified contents from memory via the DMA controller 15 and generates memory reference values through an authentication engine 20 for desired memory location or blocks. Typical authentication schemes include, but are not limited to, secure hash algorithm 1 (SHA-1) and message digest algorithm 5 (MD5). The memory reference value represents a distinct value for specific memory contents. Each memory reference value is stored in the memory reference file 40 for later comparison. The processes 115 and 117 are repeated until the autonomous memory checker 10 receives another signal from the host to enter a runtime check mode 120.

[0025] In runtime check mode 120, an adjustable time randomizer initiates memory reference comparison 130. When timer module 35 wakes up the rest of the circuitry in autonomous memory checker 10, controller 25 signals DMA controller 15 to fetch memory contents from specified blocks and addresses. The fetched memory contents are fed into the authentication engine 20 to generate a runtime reference value. The runtime reference value is compared with the stored value in memory reference file 40. The compared runtime and memory reference values correspond to identical memory blocks and addresses. If the runtime and memory reference values match, then processes 130 and 140 are repeated whenever the adjustable time randomizer initiates another memory reference comparison. If the memory reference values do not match, then autonomous memory checker 10 enters an error mode 155 and hardware actions are taken.

[0026] Once an error occurs, autonomous memory checker 10 has several hardware actions that can be taken. One possible action is entering a hardwired, reduced system state. This wire would bring the functionality of the system to a non-secure state such as E-911 for cellular phones. An external alert is another method of a hardware action for informing error. This could be something as simple as an LED indicating that the integrity of memory content of the device cannot be trusted. For extremely sensitive data, the hardware actions can also include automatic clearing of memory containing sensitive information.

[0027] FIG 3 is a system block diagram showing an autonomous memory checker 205 coupled to a host processor 210. Autonomous memory checker 205 has a master bus 245 which is used to read data from an internal memory block 215 and external memory blocks 220 and 225. Autonomous memory checker 205 is programmed by the host processor 210

over slave bus 255. Autonomous memory checker 205 has a DMA capability and does not require any actions from host processor 210 once it has been programmed. If the write-read checks are implemented, autonomous memory checker 205 should also be able to write to memory blocks 215, 220, and 225.

[0028] Autonomous memory checker 205 also has the capability to monitor contents of registers in peripheral devices 260 and 265. During the boot time, autonomous memory checker 205 can authenticate contents of device registers and store register reference values in its memory reference file 40. Then, autonomous memory checker 205 can compare real-time register values in devices 260 and 265 with its stored values during runtime.

[0029] FIG 4 is a second flow chart 300 displaying operating steps of autonomous memory checker 205. In an embodiment of the system, autonomous memory checker 205 is implemented with a block-check scheme, using hash functions to generate memory reference values. First, host processor 210 loads address and length pairs for memory blocks that need to be hashed, as shown in a procedure box 310. Host processor 210 then sets single hash memory enables in control register and "hash once" bit in command register, as shown in a procedure box 320.

[0030] At this stage, autonomous memory checker 205 enters the initial memory reference mode, as previously explained in FIG. 2. Autonomous memory checker 205 hashes memory, places a hash value in the hash register file, and generates "done" interrupt, as shown in a procedure box 330. Then, host processor 210 reads generated hash value from the autonomous memory checker 205 and verifies the digital signature of the memory, as shown in the procedure box 340. If the digital signature is incorrect, then there is a code integrity error, as shown in a decision box 350. If the values match, host processor 210 tells autonomous memory checker 205 whether a memory block comparison should be active during runtime, as shown in a decision box 360. Otherwise, memory block comparison will be skipped during runtime code hashing as indicated in the NO decision of box 360. If host processor 210 wants autonomous memory checker 205 to proceed with block comparisons, host processor 210 will set runtime hash memory enables in the control register and "Run Time Hash" bit in the command register, as shown in a procedure 370.

[0031] At this point, autonomous memory checker 205 cannot be turned off and operates independent of commands from the host processor 210. Autonomous memory checker 205

frequently compares hash results from memory blocks 215, 220, and 225 during runtime with stored values in the hash register file. The device operates in an infinite loop 380 until a memory reference mismatch occurs. The check is performed periodically or randomly. A random approach makes an attack such as a piggy back attack where a memory or memory content is swapped out more difficult because the checks do not occur in any regular timed sequence.

[0032] While at least one exemplary embodiment has been presented in the foregoing detailed description, it should be appreciated that a vast number of variations exist. It should also be appreciated that the exemplary embodiment or exemplary embodiments are only examples, and are not intended to limit the scope, applicability, or configuration of the invention in any way. Rather, the foregoing detailed description will provide those skilled in the art with a convenient road map for implementing the exemplary embodiment or exemplary embodiments. It should be understood that various changes can be made in the function and arrangement of elements without departing from the scope of the invention as set forth in the appended claims and the legal equivalents thereof.